Kotlin - Functions

Kotlin is a statically typed language, hence, functions play a great role in it. We are pretty familiar with function, as we are using function throughout our examples in our last chapters. A **function** is a block of code which is written to perform a particular task. Functions are supported by all the modern programming languages and they are also known as **methods** or **subroutines**.

At a broad level, a function takes some input which is called **parameters**, perform certain actions on these inputs and finally returns a value.

Kotlin Built-in Functions

Kotlin provides a number of built-in functions, we have used a number of buil-in functions in our examples. For example **print()** and **println()** are the most commonly used built-in function which we use to print an output to the screen.

Example

```
fun main(args: Array<String>) {
   println("Hello, World!")
}
```

User-Defined Functions

Kotlin allows us to create our own function using the keyword **fun**. A user defined function takes one or more parameters, perform an action and return the result of that action as a value.

Syntax

```
fun functionName(){
   // body of function
}
```

Once we defined a function, we can call it any number of times whenever it is needed. Following isa simple syntax to call a Kotlin function:

```
functionName()
```

Example

Following is an example to define and call a user-defined function which will print simple "Hello, World!":

```
fun main(args: Array<String>) {

   printHello()

}

fun printHello(){
   println("Hello, World!")
}
```

When you run the above Kotlin program, it will generate the following output:

```
Hello, World!
```

Function Parameters

A user-defined function can take zero or more parameters. Parameters are options and can be used based on requirement. For example, our above defined function did not make use of any paramenter.

Example

Following is an example to write a user-defined function which will add two given numbers and print their sum:

```
fun main(args: Array<String>) {
   val a = 10
   val b = 20

   printSum(a, b)

}

fun printSum(a:Int, b:Int){

   println(a + b)
}
```

When you run the above Kotlin program, it will generate the following output:

```
30
```

Return Values

A kotlin function return a value based on requirement. Again it is very much optional to return a value.

To return a value, use the **return** keyword, and specify the return type after the function's parantheses

Example

Following is an example to write a user-defined function which will add two given numbers and return the sum:

```
fun main(args: Array<String>) {
   val a = 10
   val b = 20

   val result = sumTwo(a, b)
   println( result )

}

fun sumTwo(a:Int, b:Int):Int{
   val x = a + b

   return x
}
```

When you run the above Kotlin program, it will generate the following output:

```
30
```

Unit-returning Functions

If a function does not return a useful value, its return type is **Unit**. Unit is a type with only one value which is **Unit**.

```
fun sumTwo(a:Int, b:Int):Unit{
   val x = a + b

   println( x )
}
```

The Unit return type declaration is also optional. The above code is equivalent to:

```
fun sumTwo(a:Int, b:Int){
   val x = a + b

   println( x )
}
```

Kotlin Recursive Function

Recursion functions are useful in many scenerios like calculating factorial of a number or generating fibonacci series. Kotlin supports recursion which means a Kotlin function can call itself.

Syntax

```
fun functionName(){
   ...
   functionName()
   ...
}
```

Example

Following is a Kotlin program to calculate the factorial of number 10:

```
fun main(args: Array<String>) {
   val a = 4

   val result = factorial(a)
   println( result )

}

fun factorial(a:Int):Int{
   val result:Int

   if( a <= 1){
      result = a
   }else{
      result = a*factorial(a-1)
   }

   return result
}
```

When you run the above Kotlin program, it will generate the following output:

```
24
```

Kotlin Tail Recursion

A recursive function is eligible for **tail recursion** if the function call to itself is the last operation it performs.

Example

Following is a Kotlin program to calculate the factorial of number 10 using tail recursion. Here we need to ensure that the multiplication is done before the recursive call, not after.

```
fun main(args: Array<String>) {
   val a = 4

   val result = factorial(a)
```

```
    println( result )

}

fun factorial(a: Int, accum: Int = 1): Int {
    val result = a * accum
    return if (a <= 1) {
        result
    } else {
        factorial(a - 1, result)
    }
}
```

When you run the above Kotlin program, it will generate the following output:

```
24
```

> Kotlin tail recursion is useful while calculating factorial or some other processing on large numbers. So to avoide *java.lang.StackOverflowError*, you must use tail recursion.

Higher-Order Functions

A higher-order function is a function that takes another function as parameter and/or returns a function.

Example

Following is a function which takes two integer parameters, a and b and additionally, it takes another function **operation** as a parameter:

```
fun main(args: Array<String>) {

    val result = calculate(4, 5, ::sum)
    println( result )

}
fun sum(a: Int, b: Int) = a + b

fun calculate(a: Int, b: Int, operation:(Int, Int) -> Int): Int {
    return operation(a, b)
}
```

When you run the above Kotlin program, it will generate the following output:

```
9
```

Here we are calling the **higher-order function** passing in two integer values and the function argument **::sum**. Here **::** is the notation that references a function by name in Kotlin.

Example

Let's look one more example where a function returns another function. Here we defined a higher-order function that returns a function. Here **(Int) -> Int** represents the parameters and return type of the **square** function.

```
fun main(args: Array<String>) {
   val func = operation()
   println( func(4) )

}
fun square(x: Int) = x * x

fun operation(): (Int) -> Int {
    return ::square
}
```

When you run the above Kotlin program, it will generate the following output:

```
9
```

Kotlin Lambda Function

Kotlin lambda is a function which has no name and defined with a curly braces {} which takes zero or more parameters and body of function.

The body of function is written after variable (if any) followed by -> operator.

Syntax

```
{variable with type -> body of the function}
```

Example

```
fun main(args: Array<String>) {

   val upperCase = { str: String -> str.toUpperCase() }

   println( upperCase("hello, world!") )

}
```

When you run the above Kotlin program, it will generate the following output:

```
HELLO, WORLD!
```

Kotlin Inline Function

An **inline** function is declared with **inline** keyword. The use of inline function enhances the performance of higher order function. The inline function tells the compiler to copy parameters and functions to the call site.

Example

```
fun main(args: Array<String>) {

   myFunction({println("Inline function parameter")})

}
inline fun myFunction(function:()-> Unit){
   println("I am inline function - A")

   function()

   println("I am inline function - B")
}
```

When you run the above Kotlin program, it will generate the following output:

```
I am inline function - A
Inline function parameter
I am inline function - B
```

Quiz Time (Interview & Exams Preparation)

Q 1 - Which of the following is true about Kotlin functions?

A - Kotlin functions can take one or more parameters

B - Kotlin functions can be recursive

C - Kotlin functions can return values.

D - All of the above

Q 2 - What will be the output of the following program:

```
fun main(args: Array<String>) {
   val a = 100000

   val result = factorial(a)
   println( result )

}

fun factorial(a:Int):Int{
   val result:Int

   if( a <= 1){
      result = a
```

```
    }else{
       result = a*factorial(a-1)
    }

    return result
 }
```

A - This will print 0

B - This will raise just a warning

C - Compilation will stop with error

D - None of the above

Q 2 - What is Tail Recursion?

A - Calculations are performed first, then recursive calls are executed.

B - Tail Recursion avoids the risk of stack overflow

C - Recursive call passes the result of current step to the next recursive call

D - All of the above